M 1.1:
Running the program gave the desired output of 86.73% accuracy and using time, we measured the elapsed time of the whole python program to be 24.645 seconds.

M 1.2:
The program gave the same output accuracy of 86.73% with an elapsed time of 46.531 seconds.

M 1.3:
The most time consuming kernels are cudaStreamCreateWithFlags, which uses 46.64% of the total time, cudaFree, which uses 28.60% of the total time, and cudaMemGetInfo, which uses 20.87% of the total time.

M 2.1:
With the ece408-high model with a data size of 10000, the model has a correctness of 85.62% with an op time of 11.827049 and a real time of 57.458 seconds. For the ece408-low model with a data size of 10000, the model has a correctness of 62.9% with an op time of 12.185129 and a real time of 55.375 seconds.

Contributions:
Rvarma2: Writing the code for new-forward.h
Stang17: Writing the report

M 3.1:
With the ece408-high model with a data size of 10,000, the model had a correctness of 85.62% with an op time of 0.5585 seconds and a real time of 40.935 seconds. From the nvprof, the forward_kernel took 558.45ms. For the ece408-low model, the model had a correctness of 62.9% with an op time of 0.559 seconds and a real time of 1 minute and 4.698 seconds. The forward_kernel took 559.03ms to complete.

Contributions:
Rvarma2: Writing and debugging code for new-forward.cuh
stang17: Writing code for new-forwawrd.cuh and writing report

Optimization Approach and Results:

At first, we decided to use the textbook code for the basic convolution implementation. With the code, we launched blocks with a dimension size of 16x16x1 and grids with a dimension of BxMx4. The resulting code gave us an op time of 558.15ms and an 86.87% compute time on the function. With such a large time, we looked at issues with the code and how to improve it. The issues we identified were that there was low memory reuse and bad memory access patterns.

We fixed the issues we found previously by having the block shapes match the input feature shapes and by having each thread load an input value from the input feature map into shared memory, thereby making the reads faster. With this set up, our block dimension was 28x28x1 and the grid dimension was BxMx1. This gave us an op time of 397.24ms and an 82.26% compute time on the function. After these optimizations, we realized another issue with the convolution filter memory. Since the memory is read many times but never modified, we moved the convolution filters into constant memory. This drastically decreased our time, giving us 129.49ms and a 59.49% compute time on the function.

We then noticed that variables M, H, W, and K never change, and decided to change them into defines so that the compiler can optimize the variables instead of introducing additional computations. This decreased our op time to 82.098ms with 48.65% compute time usage on the function. Our last optimization was to utilize 1D shared memory blocks for better memory access patterns and had each block compute M outputs instead of only one. In the end, we were able to get an op time of 11.356ms and a 9.04% compute time on the function.

Some other ideas that we tried but weren't fruitful were moving the convolution filters into shared memory, moving them first into constant memory and then into shared memory, and tons of different block and grid shapes and dimensions to improve memory access patterns. One other major path that we tried was utilizing the unroll and matrix multiplication method referenced in the textbook. While doing that method, we were able to bring the function down to about 50ms, which was slower than our final solution.

Contributions:
Rvarma2: working on the direct convolution method which became our final
stang17: working on initial method and unroll + matrix multiply, writing the report